Advances in the Design and Implementation of a Multi-Tier Architecture in the GIPSY Environment

Bin Han and Serguei A. Mokhov and Joey Paquet
Department of Computer Science and Software Engineering
Concordia University, Montreal, Quebec, Canada
Email: {bin_ha,mokhov,paquet}@cse.concordia.ca

Abstract

We present advances in the software engineering design and implementation of the multi-tier run-time system for the General Intensional Programming System (GIPSY) by further unifying the distributed technologies used to implement the Demand Migration Framework (DMF) in order to streamline distributed execution of hybrid intensional-imperative programs using Java.

Keywords: intensional programming, run-time system, multi-tier architecture, General Intensional Programming System (GIPSY), General Eduction Engine (GEE), Demand Migration Framework (DMF), Demand Migration System (DMS), Jini, JMS, multi-threading, RMI, Abstract Factory pattern, Factory Method pattern, Strategy pattern, Singleton pattern

1 Introduction

Intensional programming implies a declarative programming language based on the denotational semantics [1]. The declarations are evaluated in an inherent multi-dimensional context space [32]. The GIPSY project [22, 13, 16] aims at providing a platform for the investigation on the intensional and hybrid intensional-imperative programming. The GIPSY's compiler, GIPC, is based on the notion of Generic Intensional Programming Language (GIPL) [18, 21, 23, 27], which is the core run-time language into which all other flavors of the Lucid (a family of intensional programming languages) language can be translated to. The notion of a generic language also solved the problem of language-independence of the run-time system by allowing a common representation for all compiled programs, the Generic Eduction Engine Resources (GEER). A generic distributed run-time system has been proposed in [19], this paper will present the design and implementation progress so far and the immediate future work.

1.1 Problem Statement

Due to Lucid's denotational (prescriptive) semantics, its inventors had mentioned the inherent parallelism of Lucid programs: "... the whole programs can be understood as producer-consumer networks computing in parallel. Furthermore, this operational interpretation can be used as the basis of a distributed implementation ..." [1].

Although a multi-threaded and distributed architecture using Java RMI [36] has been initially designed [14], it was not fully integrated and many of the detailed working flow needed to be

clarified. Furthermore, two more separate branches of distributed computation for GIPSY emerged – the implementations based on Jini [12] and JMS [29] of the Demand Migration Framework (DMF), which in themselves are not interoperable and their top interfaces are not exactly the same complicating the integration and unification effort thereby delaying the true parallel or distributed execution of Lucid programs in the GIPSY's implementation of the run-time system – the General Eduction Engine (GEE).

1.2 Proposed Solution

Our work follows upon and enhances on GLU's generator-worker architecture [4, 10, 5] extended to be multi-tier over the course of multiple design iterations with Java. We apply most of the high-level design work produced by Paquet [19] by constructing wrapper classes for each tier type introduced in there, specifically DGT (Demand Generator Tier), DST (Demand Store Tier), DWT (Demand Worker Tier), and the GMT (General Manager Tier). Every single GIPSY node in the said design, which usually translates to a single physical computer, that has been registered within the current GIPSY network of nodes participating in computation, can host arbitrary number of instances of each tier. Since four local and distributed computation prototypes are implemented, which are multi-threaded and RMI [14, 16], Jini [35], and JMS [26] we decided to integrate them together by applying the abstract factory, factory method and strategy design patterns [8] following extreme programming [2] and the test-driven development [7, 16, 32] methodologies aiming at constructing a framework with high extensibility and maintainability for the further iterations.

1.3 Related Work

The work presented in this paper is an evolution of the original architecture for the run-time system of the GIPSY, as hinted in [18], and briefly presented in [20]. The architecture proposed in these works was itself developed following the generator-worker architecture adopted successfully by GLU [9, 10]. Despite GLU's successful implementation, the run-time system of GLU was not as scalable and flexible as the solution that we are designing for and implementing in this paper. In addition, the communication procedures of the run-time system implemented in GLU was implemented using RPC. Our solution proposes a much more flexible approach by integrating demand migration and storage by using the Demand Migration Framework (DMF), which can be concretely instantiated using various middleware technologies, such as Jini [34, 33] and JMS [24, 25].

1.3.1 Node and Tier Properties

In a GIPSY peer-to-peer computing network of nodes and tiers we aim for the following properties:

- Demands are propagated without knowing where they will be processed or stored.
- Any tier or node can fail without the system to be fatally affected.
- Nodes and tiers can seamlessly be added or removed on the fly as computation is happening.
- Nodes and tiers can be affected at run-time to the execution of any GIPSY program, i.e. a specific node or tier could be computing demands for different programs.

The conceptual design of a GIPSY node is in Figure 1.

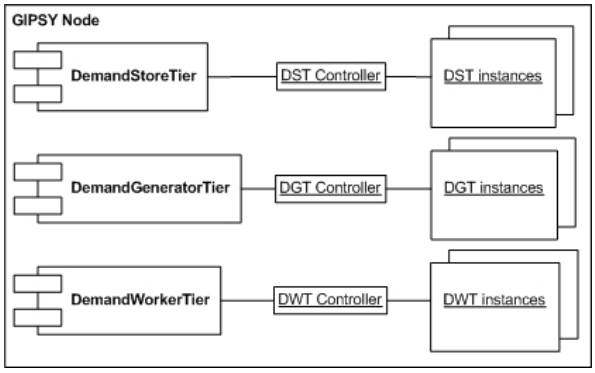

Figure 1: Design of the GIPSY Node

1.3.2 Demand Driven Computation

The central concept to this model of execution is the notion of generation, propagation, and computation of demands and their resulting values. We have demands of several types: *intensional*, *procedural*, *resource*, and *system*. Intensional demands are of the form:

```
{GEERid, programId, context}
```

where `GEERid` is a unique identifier for the GEER (i.e. the compiled program) that this demand was generated for; `programId` is an identifier declared in this GEER (in this case a Lucid identifier); and `context` is the context of evaluation of this demand. Procedural demands are of the form:

```
{GEERid, programId, Object params[], context, [code]}
```

where `GEERid` is a unique identifier for the GEER that this demand was generated for; `programId` is an identifier declared in this GEER (in this case a procedure identifier); `params[]` is an array of `Object`s that this procedure takes as arguments, `context` is the context of evaluation of this demand, and `[code]` is the (optional) executable code of the procedure, in cases where we don't want to assume that the worker has the code to be executed available. Resource demands are of the form:

```
{resourceTypeId, resourceId}
```

where `resourceTypeId` is an identifier for a resource type, which is an enumerated type now containing a GEER and possibly a procedure class. This enumerated type is extensible in order to allow for new resource types to be added later. The `resourceId` is the unique identifier for the specific resource instance being sought for by the demander (e.g. demand generator). Any new resource type created is provided with a unique identifier scheme to identify each specific resource instance of this type. System demands are of the form:

```
{destinationTierId, systemDemandTypeId, Object params[]}
```

where `destinationTierId` is the tier unique identifier of the tier to which this demand is addressed, `systemDemandTypeId` is an identifier for a system demand type, which is an enumerated type containing one element for each kind of system demand, and `params[]` is an array of `Object`s that this system demand takes as arguments, if any.

1.3.3 Demand Identifiers

Universally unique identifier The `DispatcherEntry` class uses a universally unique identifier to uniquely identify each demand within a computing GIPSY network. This identifier is local to the DMS and is invisible from the perspective of GIPSY tiers. Note that this identifier scheme generates distinct identifiers for absolutely all demands produced in the system.

Demand signature identifier This is the identifier of a demand that is generated from its tuple elements. Note that all demands generated with the same signature (e.g. {`GEER1,A,[d:2]`}) will generate the same demand signature identifier, so that the same demand generated after having been computed can be queried in the DST and its result can be extracted without recomputation, following the principles of dynamic programming.

1.3.4 Multi-Threaded and RMI Run-Time

Single-threaded, multi-threaded, and RMI-based run-time architectures with the NetCDF-supported storage for GIPSY have been explored first [14, 30, 16, 3] and an attempt of unification of them has been made. The requirement of the new multi-tier architecture states that they must be a possibility still when needed in addition to the new technologies that were further developed that are discussed in the following section.

1.3.5 Jini-DMS and JMS-DMS

To overcome GLU's inflexibility, the GIPSY was designed to include a generic and technology-independent collection of Demand Migration Systems (DMSs), which implement the Demand Migration Framework (DMF) for a particular technology or technologies to communicate and store information. Jini-DMS [35] incorporates a solution based on Jini [12] and JavaSpaces [15, 6], where Jini has been used for the design and implementation of the Transport Agents (TAs) and JavaSpaces for the design and implementation of the Demand Store. JMS-DMS [26] applied the DMF framework based on the Java Messaging Service (JMS) paradigm. The JBoss Application Server [11] has been used as JMS provider and Hypersonic Database (HSQLDB) [31], which is an embedded solution inside JBoss provides persistence and caching.

2 Design and Implementation

We present the detailed design decisions so far of our ongoing implementation here.

2.1 Multi-tier Package

Classes and interfaces for the implementation under the `gipsy.GEE.multitier` package. The corresponding wrappers are located in their respective subdirectories (sub packages). To summarize, we have a root `multitier` package:

`gipsy.GEE.multitier`

In there there are subpackages for each of the tier types and the corresponding wrapper classes among other things:

gipsy.GEE.multitier.DGT.DGTWrapper
gipsy.GEE.multitier.DST.DSTWrapper
gipsy.GEE.multitier.DWT.DWTWrapper
gipsy.GEE.multitier.GMT.GMTWrapper

2.2 Wrappers, API and Classes

Regarding the core design of the Multi-Tier Architecture and the developers' implementation efficiency, we decided to define the four aforementioned wrapper classes: `DGTWrapper` (Demand Generator Tier Wrapper), `DSTWrapper` (Demand Store Tier Wrapper), `DWTWrapper` (Demand Worker Tier Wrapper), and the `GMTWrapper` (General Manager Tier Wrapper), such that they all inherit from the same abstract class called `GenericTierWrapper` that implements the most common functionality of the interface `IMultiTierWrapper`.

The interface is to be used by any invoking application, e.g. the main class `GEE` or even some eventual external applications. We defined the preliminary API of the interface, and we will provide the code for it and adjust the tier wrapper stubs to adhere to the interface.

The interface we designed, is placed into the same package `gipsy.GEE.multitier` and is called `IMultiTierWrapper`. It is used by the `GEE` main class or the tier controller classes to hold a reference to a given tier type in a generic manner. The initial content of the interface is in Figure 1 (trimmed) that the above actual wrapper classes implement. Thus, we define the actual Java syntax interface and its implementation within the concrete wrapper classes.

```
import gipsy.Configuration;

public interface IMultiTierWrapper
extends Runnable
{
    ...
    startTier();
    stopTier();
    setConfiguration(Configuration);
    Configuration getConfiguration();
    ...
}
```

Listing 1: Primary API of the `IMultiTierWrapper` Interface.

2.3 Generic Wrapper

Two GIPSY objects included as data members in the `GenericTierWrapper` adhere to the APIs of `Configuration` and `ITransportAgent`, and are described further. All wrappers are to have a configuration instance and potentially communicate through a given transport agent (TA) implementation.

- `Configuration` contains a `Serializable` configuration of this GIPSY instance and its components, for static and run-time configuration management.
- A TA reference abstracted by the `ITransportAgent` unification interface for all transport agents (TAs) implemented in the extended DMF (Demand Migration Framework) and DMS

(Demand Migration System) and beyond. All TAs must implement this interface. This is a super-interface for the use by the engine and the multi-tier architecture. The original implementations based on Jini and JMS did not have a common super-interface, which we had to define and provide ourselves during the course of this work. After defining a family of interfaces, we can encapsulate each implementation and make them interchangeable. The strategy pattern lets the implementation technique vary independently from clients that use it.

2.4 DGT and DWT Wrappers

Both `DGTWrapper` and `DWTWrapper` classes have the common `oGEERPool` data member, which is a collection of objects of type `GIPSYProgram`, which act like local caches of the downloaded programs from the DST for execution. These are the brief descriptions of the contained components:

- `GIPSYProgram`: A dictionary of identifiers and the abstract syntax tree (AST) compiled from the program by the compiler, GIPC, also mentioned as a GEER (General Eduction Engine Resources).
- `GEERPool`: Maintains a collection of GEERs, serving as a cache in each DWT and DGT. Whenever a demand is needed pertaining to an identifier embedded in a particular GEER, DWT and DGT will search for this GEER in the local GEER pool, and if it's not cached, a resource demand is made in order to get the required GEER from elsewhere.

2.5 DST Wrapper

The `DSTWrapper` is similar to the two earlier described classes, except when it inherits from the base class `GenericTierWrapper`, the `DSTWrapper` encapsulates `oStorageSubsystem`, which is an object of type `IVWInterface`.

- `IVWInterface` is an integrated Intensional Value Warehouse, that now we refer to as a demand store (DS), specifically, if the Demand Migration System used by the DST is implemented by JMS [29], `IVWInterface` represents JBoss [11]; if implemented by Jini, represents JavaSpaces [15], if locally or by RMI, then represents NetCDF [3].

2.6 Supporting Classes

Throughout our design and development effort we further introduce a data structure, a factory method class for tiers, and the controller class to satisfy the high-level design presented earlier.

- `EDMFImplemenation` – `EDMFImplemenation` is an enumerated type containing options for multi-threaded, RMI, Jini and JMS-based communications middleware so far. It acts as an identifier for the techniques used in the implementation. In the future, others may add other types representing the new DMSs they might develop.
- `TierFactory` – `TierFactory` is an instance of the abstract factory pattern. Each tier controller (see below) maintains a reference to the subclass of `TierFactory`, which creates objects of its respective tier type. Each concrete tier factory, e.g. `DGTFactory`, provides its controller, e.g. `DGTController`, an interface to create families of DGT without specifying their concrete

implementation transport agent strategies, which enhances the flexibility and maintainability of the system.

- `NodeController` – `NodeController` is an abstract class for each tier controller. Its subclasses decide, which tier type to instantiate. Its existence facilitates further implementation and addition of any new tier types to the multi-tier architecture as needed. If necessary, there will be one controller for each tier that implements `GenericTierWrapper` running on every node, which is a physical computer. It takes responsibility of adding to (through `TierFactory`) or removing any tier instances from the node. The singleton pattern is applied here to ensure only one controller is created for each tier on each node, and the factory method pattern lets the subclasses to specify the tier objects it controls.

2.7 Design Summary

In Figure 2 is the UML diagram describing the relationships between different tier types, the configuration, TAs, the engine, and other modules we just described. This overall class diagram was abbreviate to remove the usual routine details most classes have, focusing only on the core aspects of this work.

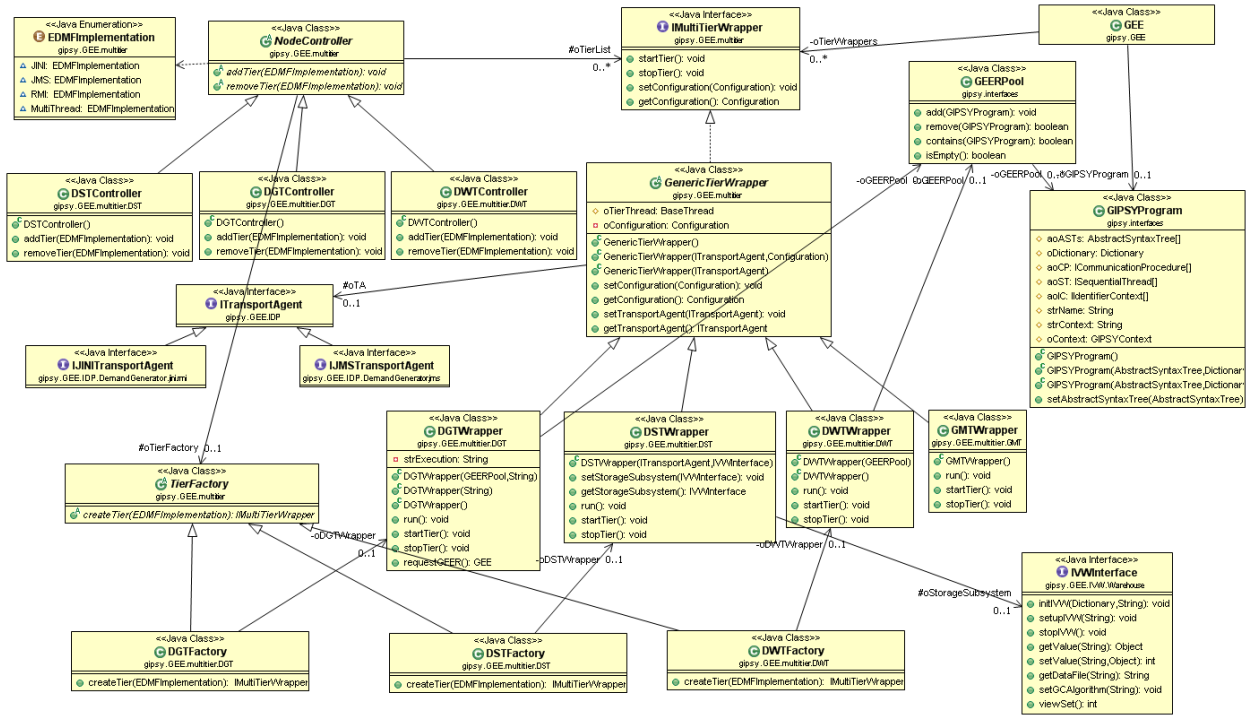

Figure 2: The Initial Multi-tier Architecture Design and Implementation

2.8 Integration with the GEE

The primary integration of the invocation of the multitier services via the main entry point of the engine, the main class `GEE`, has to be re-designed to accommodate these new developments.

The adjustments include option processing, service start-up and handling the control over the a particular tier via its controller or directly for preliminary testing purposes.

For the immediate support of the wrapper invocation we provide the means through options of `GEE.java` (the main class) to invoke a particular tier or tiers. We also create the wrapper Linux, MacOS X, and Windows shell and batch scripts to start up the tiers, and the GNU Make [28]'s `Makefile`s to build the new multitier code in Linux.

In order to start the GEE (that roughly corresponds to a GIPSY node instance) we do:

```
./gee --option
```

e.g. to start up a DGT, it is:

```
./gee --dgt[=N]
```

(where N is the optional number of instances) and the corresponding wrapper scripts, such that one can start any tier, like:

```
./dgt N
./dst N
...
```

GEE uses the newly created API in the previous section of `IMultiTierWrapper`. Based on the options, `gee` employs the factory method to instantiate the desired tier types of arbitrary number of instances, and then starts them or stops them as directed using the API of `IMultiTierWrapper`. Some of this first build interaction is under the process of migration to the `NodeController` class described earlier, i.e. `GEE` will delegate the start up activities and others to the node controller, which will in turn will use the factory pattern to manage the tiers. After having started the tiers, `GEE` proceeds to the execution of the program if there is one to execute, encapsulated into a `GIPSYProgram` instance, the GEER. As execution progresses, the demands are generated and handed off to the tiers responsible for the delivery and results computation and return along with the warehouse store caching principles described earlier.

3 Conclusion

We believe that the ongoing design and implementation presented in this work provides a feasible solution for the eductive evaluation of hybrid intensional-imperative programs and tier management. Especially, the multi-tier infrastructure, once fully implemented and tested, will offer the GIPSY run-time system high scalability and flexibility that was pending integration effort from various developers for a long time.

We had to add some extra layers of abstraction in terms of the interfaces and APIs of interfaces `IMultiTierWrapper`, `ITransportAgent`, and class `GEERPool` in order to remain extensible and flexible to accommodate any future changes to the design and implementation. We defined a new package for the multitier implementation, and described the details and the relationship of the core classes used in the design.

4 Future Work

We have listed the following short-term immediate and longer term future work items:

- Further integration of the multi-tier run-time with the rest of GEE.
- Extensive unit testing and integration testing.
- Performance testing on a hybrid network and cluster environments.
- Security layer integration and testing [17].

5 Acknowledgments

We acknowledge the reviewers of this work and their constructive feedback. This work was sponsored in part by NSERC and the Faculty of Engineering and Computer Science, Concordia University, Montreal, Canada.

References

[1] E. A. Ashcroft and W. W. Wadge. *R* for semantics. *ACM Transactions on Programming Languages and Systems*, 4(2):283–294, Apr. 1982.

[2] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2nd edition, 2005. ISBN: 0321278658.

[3] Contrubutors, University Corporation for Atmospheric Research, and National Science Foundation. NetCDF – network common data form. [online], 2003–2009. `http://www.unidata.ucar.edu/software/netcdf/`.

[4] C. Dodd. *Intensional Programming I*, chapter Rank analysis in the GLU compiler, pages 76–82. World Scientific, Singapore, 1996.

[5] A. A. Faustini and W. W. Wadge. An eductive interpreter for the language Lucid. *SIGPLAN Not.*, 22(7):86–91, 1987.

[6] R. Flenner. *Jini and JavaSpaces Application Development*. Sams, 2001.

[7] E. Gamma and K. Beck. JUnit. Object Mentor, Inc., 2001–2004. `http://junit.org/`.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN: 0201633612.

[9] R. Jagannathan and C. Dodd. GLU programmer's guide. Technical report, SRI International, Menlo Park, California, 1996.

[10] R. Jagannathan, C. Dodd, and I. Agi. GLU: A high-level system for granular data-parallel programming. In *Concurrency: Practice and Experience*, volume 1, pages 63–83, 1997.

[11] JBoss. JBoss application server guide, 2007. `http://www.jboss.org/products/jbossas`.

[12] Jini Community. *Jini Network Technology*. Sun Microsystems, Inc., Sept. 2007. `http://java.sun.com/developer/products/jini/index.jsp`.

[13] B. Lu. *Developing the Distributed Component of a Framework for Processing Intensional Programming Languages*. PhD thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Mar. 2004.

[14] B. Lu, P. Grogono, and J. Paquet. Distributed execution of multidimensional programming languages. In *Proceedings of the 15th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, volume 1, pages 284–289. International Association of Science and Technology for Development, Nov. 2003.

[15] Q. H. Mamoud. *Getting Started With JavaSpaces Technology: Beyond Conventional Distributed Programming Paradigms*. Sun Microsystems, Inc., July 2005. `http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/`.

[16] S. A. Mokhov. Towards hybrid intensional programming with JLucid, Objective Lucid, and General Imperative Compiler Framework in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Oct. 2005. ISBN 0494102934.

[17] S. A. Mokhov. Towards security hardening of scientific distributed demand-driven and pipelined computing systems. In *Proceedings of the 7th International Symposium on Parallel and Distributed Computing (ISPDC'08)*, pages 375–382, Krakow, Poland, July 2008. IEEE Computer Society.

[18] J. Paquet. *Scientific Intensional Programming*. PhD thesis, Department of Computer Science, Laval University, Sainte-Foy, Canada, 1999.

[19] J. Paquet. A multi-tier architecture for the distributed eductive execution of hybrid intensional programs. Unpublished, 2009.

[20] J. Paquet and P. Kropf. The GIPSY architecture. In *Proceedings of Distributed Computing on the Web*, Quebec City, Canada, 2000.

[21] J. Paquet, S. A. Mokhov, and X. Tong. Design and implementation of context calculus in the GIPSY environment. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 1278–1283, Turku, Finland, July 2008. IEEE Computer Society.

[22] J. Paquet and A. H. Wu. GIPSY – a platform for the investigation on intensional programming languages. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*, pages 8–14, Las Vegas, USA, June 2005. CSREA Press.

[23] J. Plaice, B. Mancilla, G. Ditu, and W. W. Wadge. Sequential demand-driven evaluation of eager TransLucid. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 1266–1271, Turku, Finland, July 2008. IEEE Computer Society.

[24] A. H. Pourteymour, E. Vassev, and J. Paquet. Towards a new demand-driven message-oriented middleware in GIPSY. In *Proceedings of PDPTA 2007*, pages 91–97, Las Vegas, USA, June 2007. PDPTA, CSREA Press.

[25] A. H. Pourteymour, E. Vassev, and J. Paquet. Design and implementation of demand migration systems in GIPSY. In *Proceedings of PDPTA 2009*, Las Vegas, USA, June 2008. CSREA Press.

[26] A. H. Pouteymour. Comparative study of Demand Migration Framework implementation using JMS and Jini. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Sept. 2008.

[27] T. Rahilly and J. Plaice. A multithreaded implementation for TransLucid. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 1272–1277, Turku, Finland, July 2008. IEEE Computer Society.

[28] R. Stallman, R. McGrath, P. Smith, and the GNU Project. GNU Make. Free Software Foundation, Inc., [online], 1997–2006. `http://www.gnu.org/software/make/`.

[29] Sun Microsystems. *Java Message Service (JMS)*. Sun Microsystems, Inc., Sept. 2007. `http://java.sun.com/products/jms/`.

[30] L. Tao. Warehouse and garbage collection in the GIPSY environment. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2004.

[31] The hsqldb Development Group. HSQLDB – lightweight 100% Java SQL database engine v.1.8.0.10. hsqldb.org, 2001–2008. `http://hsqldb.org/`.

[32] X. Tong. Design and implementation of context calculus in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Apr. 2008.

[33] E. Vassev and J. Paquet. A general architecture for demand migration in a demand-driven execution engine in a heterogeneous and distributed environment. In *Proceedings of the 3rd Annual Communication Networks and Services Research Conference (CNSR 2005)*, pages 176–182, Halifax, Nova Scotia, May 2005. IEEE.

[34] E. Vassev and J. Paquet. A generic framework for migrating demands in the GIPSY's demand-driven execution engine. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*, pages 29–35, Las Vegas, USA, June 2005. CSREA Press.

[35] E. I. Vassev. General architecture for demand migration in the GIPSY demand-driven execution engine. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, June 2005. ISBN 0494102969.

[36] A. Wollrath and J. Waldo. Java RMI tutorial. Sun Microsystems, Inc., 1995–2005. `http://java.sun.com/docs/books/tutorial/rmi/index.html`.